

The Complexity of Abstract Machines

Beniamino Accattoli

INRIA & LIX, École Polytechnique

beniamino.accattoli@inria.fr

The λ -calculus is a peculiar computational model whose definition does not come with a notion of machine. Unsurprisingly, implementations of the λ -calculus have been studied for decades. Abstract machines are implementations schema for fixed evaluation strategies that are a compromise between theory and practice: they are concrete enough to provide a notion of machine and abstract enough to avoid the many intricacies of actual implementations. There is an extensive literature about abstract machines for the λ -calculus, and yet—quite mysteriously—the efficiency of these machines with respect to the strategy that they implement has almost never been studied.

This paper provides an unusual introduction to abstract machines, based on the complexity of their overhead with respect to the length of the implemented strategies. It is conceived to be a tutorial, focusing on the case study of implementing the weak head (call-by-name) strategy, and yet it is an original re-elaboration of known results. Moreover, some of the observation contained here never appeared in print before.

1 Cost Models & Size-Expllosion

The λ -calculus is an undeniably elegant computational model. Its definition is given by three constructors and only one computational rule, and yet it is Turing-complete. A charming feature is that it does not rest on any notion of machine or automaton. The catch, however, is that its cost model are far from being evident. What should be taken as time and space measures for the λ -calculus? The natural answers are the number of computational steps (for time) and the maximum size of the terms involved in a computation (for space). Everyone having played with the λ -calculus would immediately point out a problem: the λ -calculus is a nondeterministic system where the number of steps depends much on the evaluation strategy, so much that some strategies may diverge when others provide a result (but fortunately the result, if any, does not depend on the strategy). While this is certainly an issue to address, it is not the serious one. The big deal is called *size-explosion*, and it affects all evaluation strategies.

Size-Expllosion. There are families of terms where the size of the n -th term is linear in n , evaluation takes a linear number of steps, but the size of the result is exponential in n . Therefore, the number of steps does not even account for the time to write down the result, and thus at first sight it does not look as a reasonable cost model. Let's see examples.

The simplest one is a variation over the famous looping λ -term $\Omega := (\lambda x.xx)(\lambda x.xx) \rightarrow_{\beta} \Omega \rightarrow_{\beta} \dots$. In Ω there is an infinite sequence of duplications. In the first size-exploding family there is a sequence of n nested duplications. We define both the family $\{t_n\}_{n \in \mathbb{N}}$ of size-exploding terms and the family $\{u_n\}_{n \in \mathbb{N}}$ of results of the evaluation

$$\begin{array}{lll} t_0 & := & y \\ t_{n+1} & := & (\lambda x.xx)t_n \end{array} \qquad \begin{array}{lll} u_0 & := & y \\ u_{n+1} & := & u_n u_n \end{array}$$

We use $|t|$ for the size of a term, *i.e.* the number of symbols to write it, and say that a term is *neutral* if it is normal and it is not an abstraction.

Proposition 1.1 (Open and Rightmost-Innermost Size-Explosion). *Let $n \in \mathbb{N}$. Then $t_n \rightarrow_\beta^n u_n$, moreover $|t_n| = O(n)$, $|u_n| = \Omega(2^n)$, and u_n is neutral.*

Proof. By induction on n . The base case is immediate. The inductive case: $t_{n+1} = (\lambda x.xx)t_n \rightarrow_\beta^n (\lambda x.xx)u_n \rightarrow_\beta u_n u_n = u_{n+1}$, where the first sequence is obtained by the *i.h.* The bounds on the sizes are immediate, as well as the fact that u_{n+1} is neutral. \square

Strategy-Independent Size-Explosion. The example relies on rightmost-innermost evaluation (*i.e.* the strategy that repeatedly selects the rightmost-innermost β -redex) and open terms (the free variable $t_0 = y$). In fact, evaluating the same family in a leftmost-outermost way would produce an exponentially long evaluation sequence. One may then believe that size-explosion is a by-product of a clumsy choice for the evaluation strategy. Unfortunately, this is not the case. It is not hard to modify the example as to make it strategy-independent, and it is also easy to get rid of open terms. Let the identity combinator be $I := \lambda z.z$ (it can in fact be replaced by any closed abstraction). Define

$$\begin{array}{ll} r_1 := \lambda x.\lambda y.(yxx) & p_0 := I \\ r_{n+1} := \lambda x.(r_n(\lambda y.(yxx))) & p_{n+1} := \lambda y.(yp_np_n) \end{array}$$

The size-exploding family is $\{r_n I\}_{n \in \mathbb{N}}$, *i.e.* it is obtained by applying r_n to the identity $I = p_0$. The statement we are going to prove is in fact more general, about $r_n p_m$ instead of just $r_n I$, in order to obtain a simple inductive proof.

Proposition 1.2 (Closed and Strategy-Independent Size-Explosion). *Let $n > 0$. Then $r_n p_m \rightarrow_\beta^n p_{n+m}$, and in particular $r_n I \rightarrow_\beta^n p_n$. Moreover, $|r_n I| = O(n)$, $|p_n| = \Omega(2^n)$, $r_n I$ is closed, and p_n is normal.*

Proof. By induction on n . The base case: $r_1 p_m = \lambda x.\lambda y.(yxx)p_m \rightarrow_\beta (\lambda y.(yp_mp_m)) = p_{m+1}$. The inductive case: $r_{n+1} p_m = \lambda x.(r_n(\lambda y.(yxx)))p_m \rightarrow_\beta r_n(\lambda y.(yp_mp_m)) = r_n p_{m+1} \rightarrow_\beta^n p_{n+m+1}$, where the second sequence is obtained by the *i.h.* The rest of the statement is immediate. \square

The family $\{r_n I\}_{n \in \mathbb{N}}$ is interesting because no matter how one looks at it, it always explodes: if evaluation is weak (*i.e.* it does not go under abstraction) there is only one possible derivation to normal form and if it is strong (*i.e.* unrestricted) all derivations have the same length (and are permutatively equivalent). To our knowledge this family never appeared in print before.

2 The λ -Calculus is Reasonable, Indeed

Surprisingly, the isolation and the systematic study of the size-explosion problem is quite recent—there is no trace of it in the classic books on the λ -calculus, nor in any course notes we are aware of. Its essence, nonetheless, has been widespread folklore for a long time: in practice, functional languages never implement full β -reduction, considered a costly operation, and theoretically the λ -calculus is usually considered a model not suited for complexity analyses.

A way out of the issue of cost models for the λ -calculus, at first sight, is to take the time and space required for the execution of a λ -term in a fixed implementation. There is however no canonical implementation. The design of an implementation in fact rests on a number of choices. Consequently, there are a number of different but more or less equivalent machines taking a different number of steps and using different amounts of space to evaluate a term. Fixing one of them would be arbitrary, and, most importantly, would betray the machine-independent spirit of the λ -calculus.

Micro-Step Operational Semantics. Luckily, the size-explosion problem can be solved in a machine-independent way. Somewhat counterintuitively, in fact, the number of β -steps can be taken as a reasonable cost model. The basic idea is simple: one has to step out of the λ -calculus, by switching to a different setting that *mimics* β -reduction without literally doing it, acting on *compact representations* of terms to avoid size-explosion. Essentially, the recipe requires four ingredients:

1. *Statics*: λ -terms are refined with a form of *sharing* of subterms;
2. *Dynamics*: evaluation has to manipulate terms with sharing via *micro-operations*;
3. *Cost*: these micro-step operations have constant cost;
4. *Result*: micro-evaluation stops on a *shared representation of the result*.

The recipe leaves also some space for improvisation: λ -calculus can in fact be enriched with *first-class sharing* in various ways. Mainly, there are three approaches: *abstract machines*, *explicit substitutions*, and *graph rewriting*. They differ in the details but not in the essence—they can be grouped together under the slogan *micro-step operational semantics*.

Reasonable Strategies. An evaluation strategy \rightarrow for the λ -calculus is *reasonable* if there is a micro-step operational semantics M mimicking \rightarrow and such that the number of micro-steps to evaluate a term t is polynomial in the number of \rightarrow -steps to evaluate t (and in the size of t , we will come back to this point later on). If a strategy \rightarrow is reasonable then its length is a reasonable cost model, despite size-explosion: the idea is that the λ -calculus is kept as an *abstract* model, easy to define and reason about, while complexity-concerned evaluation is meant to be performed at the more sophisticated micro-step level, where the explosion cannot happen.

Of course, the design of a reasonable micro-step operational semantics depends much on the strategy and the chosen flavor of micro-steps semantics, and it can be far from easy. For *weak* strategies—used to model functional programming languages—reasonable micro-steps semantics are based on a simple form of sharing. The first result about reasonable strategies was obtained by Blelloch and Greiner in 1995 [11] and concerns indeed a weak strategy, namely the call-by-value one. At the micro-step level it relies on abstract machines. Similar results were then proved again, independently, by Sands, Gustavsson, and Moran in 2002 [13] and by Dal Lago and Martini in 2006 [12]. For *strong* strategies—at work in proof assistant engines—quite more effort and care are required. A sophisticated second-level of sharing, called *useful sharing*, is necessary to obtain reasonable micro-step semantics for strong evaluation. The first such semantics has been introduced by Accattoli and Dal Lago in 2014 [10] for the leftmost-outermost strategy, and its study is still ongoing [7, 2].

The Complexity of Abstract Machines. To sum up, various techniques, among which abstract machines, can be used to prove that the number of β -steps is a reasonable time cost model, *i.e.* a metric for time complexity. The study can then be reversed, exploring how to use this metric to study the relative complexity of abstract machines, that is, the complexity of the overhead of the machine with respect to the number of β -steps. Such a study leads to a new quantitative theory of abstract machines, where machines can be compared and the value of different design choices can be measured. The rest of the paper provides a gentle introduction to the basic concepts of the new complexity-aware theory of abstract machines being developed by the author in joint works [3, 6, 4, 7, 2] with Damiano Mazza, Pablo Barenbaum, and Claudio Sacerdoti Coen, and resting on tools and concepts developed beforehand in collaborations with Delia Kesner [9] and Ugo Dal Lago [8], as well as Kesner plus Eduardo Bonelli and Carlos Lombardi [5].

Case Study: Weak Head Strategy. The paper focuses on a case study, the weak head (call-by-name) strategy, also known as weak head reduction (we use *reduction* and *strategy* as synonymous, and prefer *strategy*), and defined as follows:

$$\frac{}{(\lambda x.t)u \rightarrow_{wh} t\{x \leftarrow u\}} \text{(root } \beta\text{)} \quad \frac{t \rightarrow_{wh} u}{tr \rightarrow_{wh} ur} \text{ (@1)} \quad (1)$$

This is probably the simplest possible evaluation strategy. Of course, it is deterministic. Let us mention two other ways of defining it, as they will be useful in the sequel. First, the given inductive definition can be unfolded into a single synthetic rule $(\lambda x.t)ur_1 \dots r_k \rightarrow_{wh} t\{x \leftarrow u\}r_1 \dots r_k$. Second, the strategy can be given via evaluation contexts: define $E := \langle \cdot \rangle \mid Er$ and define \rightarrow_{wh} as $E\langle (\lambda x.t)u \rangle \rightarrow_{wh} E\langle t\{x \leftarrow u\} \rangle$ (where $E\langle t \rangle$ is the operation of plugging t in the context E , consisting in replacing the hole $\langle \cdot \rangle$ with t).

Sometimes, to stress the modularity of the reasoning, we will abstract the weak head strategy into a generic strategy \rightarrow . Last, a *derivation* is a possibly empty sequence of rewriting steps.

3 Introducing Abstract Machines

Tasks of Abstract Machines. An abstract machine is an implementation schema for an evaluation strategy \rightarrow with sufficiently atomic operations and without too many details. A machine for \rightarrow accounts for 3 tasks:

1. *Search*: searching for \rightarrow -redexes;
2. *Substitution*: replace meta-level substitution with an approximation based on sharing;
3. *Names*: take care of α -equivalence.

Dissecting Abstract Machines. To guide the reader through the different concepts to design and analyze abstract machines, the next two subsections describe in detail two toy machines that address in isolation the first two mentioned tasks, *search* and *substitution*. They will then be merged into the Milner Abstract Machine (MAM). In Sect. 7 we will analyze the complexity of the MAM. Next, we will address *names* and describe the Krivine Abstract Machine, and quickly study its complexity.

Abstract Machines Glossary.

- An abstract machine M is given by *states*, noted s , and *transitions* between them, noted \rightsquigarrow ;
- A state is given by the *code under evaluation* plus some *data-structures* to implement *search* and *substitution*, and to take care of *names*;
- The code under evaluation, as well as the other pieces of code scattered in the data-structures, are λ -terms *not considered modulo α -equivalence*;
- Codes are over-lined, to stress the different treatment of α -equivalence;
- A code \bar{t} is *well-named* if x may occur only in \bar{u} (if at all) for every sub-code $\lambda x.\bar{u}$ of \bar{t} ;
- A state s is *initial* if its code is well-named and its data-structures are empty;
- Therefore, there is a bijection \cdot° (up to α) between terms and initial states, called *compilation*, sending a term t on the initial state t° on a well-named code α -equivalent to t ;
- An *execution* is a (potentially empty) sequence of transitions $s' \rightsquigarrow^* s$ from an initial state s' obtained by compiling a(n initial) term t_0 ;

- A state s is *reachable* if it can be obtained as the end state of an execution;
- A state s is *final* if it is reachable and no transitions apply to s .
- A machine comes with a map $\underline{\cdot}$ from states to terms, called *decoding*, that on initial states is the inverse (up to α) of compilation;
- A machine M has a set of β -*transitions* that are meant to be mapped to β -redexes (and whose name involves β) by the decoding, while the remaining *overhead transitions* are mapped on equalities;
- We use $|\rho|$ for the length of an execution ρ , and $|\rho|_\beta$ for the number of β -transitions in ρ .

Implementations. For every machine one has to prove that it correctly implements the strategy it was conceived for. Our notion, tuned towards complexity analyses, requires a perfect match between the number of β -steps of the strategy and the number of β -transitions of the machine execution.

Definition 3.1 (Machine Implementation). *A machine M implements a strategy \rightarrow on λ -terms when given a λ -term t the following holds*

1. Executions to Derivations: *for any M -execution $\rho : t^\circ \rightsquigarrow_M^* s$ there exists a \rightarrow -derivation $d : t \rightarrow^* \underline{s}$.*
2. Derivations to Executions: *for every \rightarrow -derivation $d : t \rightarrow_{wh}^* u$ there exists a M -execution $\rho : t^\circ \rightsquigarrow_M^* s$ such that $\underline{s} = u$.*
3. β -Matching: *in both previous points the number $|\rho|_\beta$ of β -transitions in ρ is exactly the length $|d|$ of the derivation d , i.e. $|d| = |\rho|_\beta$.*

Note that if a machine implements a strategy than the two are *weakly bisimilar*, where weakness is given by the fact that overhead transitions do not have an equivalent on the calculus (hence their name). Let us point out, moreover, that the β -matching requirement in our notion of implementation is unusual but perfectly reasonable, as all abstract machines we are aware of do satisfy it.

4 The Searching Abstract Machine

Strategies are usually specified through inductive rules as those in (1). The inductive rules incorporate in the definition the search for the next redex to reduce. Abstract machines make such a search explicit and actually ensure two related subtasks:

1. Store the current evaluation context in appropriate *data-structures*.
2. Search *incrementally*, exploiting previous searches.

For weak head reduction the search mechanism is basic. The data structure is simply a stack π storing the arguments of the current head subterm.

Searching Abstract Machine. The searching abstract machine (Searching AM) in Fig. 1 has two components, the *code* in evaluation position and the *argument stack*. The machine has only two transitions, corresponding to the rules in (1), one β -transition ($\rightsquigarrow_{r\beta}$) dealing with β -redexes in evaluation position and one overhead transition ($\rightsquigarrow_{@l}$) adding a term on the argument stack. Compilation of a (well-named) term t into a machine state simply sends t to the *initial state* (\bar{t}, ε) . The decoding given in Fig. 1 is defined inductively on the structure of states. It can equivalently be given contextually, by associating an evaluation context to the data structures—in our case sending the argument stack π to a context $\underline{\pi}$ by setting $\underline{\varepsilon} := \langle \cdot \rangle$, $\underline{u :: \pi} := \underline{\pi} \langle \langle \cdot \rangle u \rangle$, and $(\bar{t}, \pi) := \underline{\pi} \langle t \rangle$. It is useful to have both definitions since sometimes one is more convenient than the other.

$\begin{array}{lcl} \text{Stacks} & \pi & := \varepsilon \mid \bar{t} :: \pi \\ \text{Compilation} & t^\circ & := (\bar{t}, \varepsilon) \end{array}$	$\begin{array}{lcl} \text{Decoding} & (\bar{t}, \varepsilon) & := t \\ & (\bar{t}, \bar{u} :: \pi) & := (\bar{t}\bar{u}, \pi) \end{array}$															
<table border="1" style="margin: auto; border-collapse: collapse; width: fit-content;"> <thead> <tr> <th style="padding: 2px;">Code</th> <th style="padding: 2px;">Stack</th> <th style="padding: 2px;">Trans.</th> <th style="padding: 2px;">Code</th> <th style="padding: 2px;">Stack</th> </tr> </thead> <tbody> <tr> <td style="padding: 2px;">$\bar{t}\bar{u}$</td><td style="padding: 2px;">π</td><td style="padding: 2px;">$\rightsquigarrow_{@l}$</td><td style="padding: 2px;">\bar{t}</td><td style="padding: 2px;">$\bar{u} :: \pi$</td></tr> <tr> <td style="padding: 2px;">$\lambda x.\bar{t}$</td><td style="padding: 2px;">$\bar{u} :: \pi$</td><td style="padding: 2px;">$\rightsquigarrow_{r\beta}$</td><td style="padding: 2px;">$\bar{t}\{x \leftarrow \bar{u}\}$</td><td style="padding: 2px;">π</td></tr> </tbody> </table>		Code	Stack	Trans.	Code	Stack	$\bar{t}\bar{u}$	π	$\rightsquigarrow_{@l}$	\bar{t}	$\bar{u} :: \pi$	$\lambda x.\bar{t}$	$\bar{u} :: \pi$	$\rightsquigarrow_{r\beta}$	$\bar{t}\{x \leftarrow \bar{u}\}$	π
Code	Stack	Trans.	Code	Stack												
$\bar{t}\bar{u}$	π	$\rightsquigarrow_{@l}$	\bar{t}	$\bar{u} :: \pi$												
$\lambda x.\bar{t}$	$\bar{u} :: \pi$	$\rightsquigarrow_{r\beta}$	$\bar{t}\{x \leftarrow \bar{u}\}$	π												

Figure 1: Searching Abstract Machine (Searching AM).

Implementation. We now show the implementation theorem for the Searching AM with respect to the weak head strategy. Despite the simplicity of the machine, we provide a quite accurate account of the proof of the theorem, to be taken as a modular recipe. The proofs of the other implementation theorems in the paper will then be omitted as they follow exactly the same structure, *mutatis mutandis*.

The *executions-to-derivations* part of the implementation theorem always rests on a lemma about the decoding of transitions, that in our case takes the following form.

Lemma 4.1 (Transitions Decoding). *Let s be a Searching AM state.*

1. β -Transition: if $s \rightsquigarrow_{r\beta} s'$ then $\underline{s} \rightarrow_{\beta} \underline{s}'$.
2. Overhead Transition: if $s \rightsquigarrow_{@l} s'$ then $\underline{s} = \underline{s}'$.

Proof. The first point is more easily proved using the contextual definition of decoding.

1. $\underline{s} = (\lambda x.\bar{t}, \bar{u} :: \pi) = \bar{u} :: \pi \langle \lambda x.t \rangle = \pi \langle (\lambda x.t)u \rangle \rightarrow_{\beta} \pi \langle t\{x \leftarrow u\} \rangle = \underline{s}'$.
2. $\underline{s}' = (\bar{t}, \bar{u} :: \pi) = (\bar{t}\bar{u}, \pi) = \underline{s}$. □

Transitions decoding extends to a projection of executions to derivations (via a straightforward induction on the length of the execution), as required by the implementation theorem. For the *derivations-to-executions* part of the theorem, we proceed similarly, by first proving that single weak head steps are simulated by the Searching AM and then extending the simulation to derivations via an easy induction. There is a subtlety, however, because, if done naively, one-step simulations do not compose.

Let us explain the point. Given a step $t \rightarrow_{wh} u$ there exists a state s such that $t^\circ \rightsquigarrow_{@l}^* \rightsquigarrow_{r\beta} s$ and $\underline{s} = u$, as expected. This property, however, cannot be iterated to build a many-steps simulation, because $\underline{s} = u$ does not imply $s = u^\circ$, i.e. s in general is not the compilation of u . To make things work, the simulation of $t \rightarrow_{wh} u$ should not start from t° but from a state s' such that $\underline{s}' = t$. Now, the proof of the step simulation lemma we just described relies on the following three properties:

Lemma 4.2 (Bricks for Step Simulation).

1. Vanishing Transitions Terminate: $\rightsquigarrow_{@l}$ terminates;
2. Determinism: the Searching AM is deterministic;
3. Progress: final Searching AM states decode to \rightarrow_{wh} -normal terms.

Proof. *Termination:* $\rightsquigarrow_{@l}$ -sequences are bound by the size of the code. *Determinism:* $\rightsquigarrow_{r\beta}$ and $\rightsquigarrow_{@l}$ clearly do not overlap and can be applied in a unique way. *Progress:* final states have the form $(\lambda x.\bar{t}, \varepsilon)$ and (x, π) , that both decode to \rightarrow_{wh} -normal forms. □

Environments $E := \varepsilon \mid [x \leftarrow \bar{t}] :: E$ Compilation $t^\circ := (\bar{t}, \varepsilon)$	Decoding $\frac{(\bar{t}, \varepsilon)}{(\bar{t}, [x \leftarrow \bar{u}] :: E)} := \underline{t}$ $\frac{(\bar{t}, [x \leftarrow \bar{u}] :: E)}{(\bar{t} \{ x \leftarrow \bar{u} \}, E)} := \underline{(\bar{t} \{ x \leftarrow \bar{u} \}, E)}$														
<table border="1" style="margin: auto; border-collapse: collapse;"> <thead> <tr> <th style="padding: 2px;">Code</th> <th style="padding: 2px;">Env</th> <th style="padding: 2px;">Trans</th> <th style="padding: 2px;">Code</th> <th style="padding: 2px;">Env</th> </tr> </thead> <tbody> <tr> <td style="padding: 2px;">$(\lambda x.\bar{t})\bar{u}\bar{r}_1 \dots \bar{r}_k$</td> <td style="padding: 2px;">E</td> <td style="padding: 2px; border-bottom: 1px solid black;">$\rightsquigarrow_{d\beta}$</td> <td style="padding: 2px;">$\bar{t}\bar{r}_1 \dots \bar{r}_k$</td> <td style="padding: 2px;">$[x \leftarrow \bar{u}] :: E$</td> </tr> <tr> <td style="padding: 2px;">$x\bar{r}_1 \dots \bar{r}_k$</td> <td style="padding: 2px;">$E :: [x \leftarrow \bar{t}] :: E'$</td> <td style="padding: 2px; border-bottom: 1px solid black;">\rightsquigarrow_{var}</td> <td style="padding: 2px;">$\bar{t}^\alpha \bar{r}_1 \dots \bar{r}_k$</td> <td style="padding: 2px;">$E :: [x \leftarrow \bar{t}] :: E'$</td> </tr> </tbody> </table> <p style="margin-top: 2px;">where \bar{t}^α denotes \bar{t} where bound names have been freshly renamed.</p>	Code	Env	Trans	Code	Env	$(\lambda x.\bar{t})\bar{u}\bar{r}_1 \dots \bar{r}_k$	E	$\rightsquigarrow_{d\beta}$	$\bar{t}\bar{r}_1 \dots \bar{r}_k$	$[x \leftarrow \bar{u}] :: E$	$x\bar{r}_1 \dots \bar{r}_k$	$E :: [x \leftarrow \bar{t}] :: E'$	\rightsquigarrow_{var}	$\bar{t}^\alpha \bar{r}_1 \dots \bar{r}_k$	$E :: [x \leftarrow \bar{t}] :: E'$
Code	Env	Trans	Code	Env											
$(\lambda x.\bar{t})\bar{u}\bar{r}_1 \dots \bar{r}_k$	E	$\rightsquigarrow_{d\beta}$	$\bar{t}\bar{r}_1 \dots \bar{r}_k$	$[x \leftarrow \bar{u}] :: E$											
$x\bar{r}_1 \dots \bar{r}_k$	$E :: [x \leftarrow \bar{t}] :: E'$	\rightsquigarrow_{var}	$\bar{t}^\alpha \bar{r}_1 \dots \bar{r}_k$	$E :: [x \leftarrow \bar{t}] :: E'$											

Figure 2: Micro-Substituting Abstract Machine (Micro AM).

Lemma 4.3 (One-Step Simulation). *Let s be a Searching AM state. If $\underline{s} \rightarrow_{wh} u$ then there exists a state s' such that $s \rightsquigarrow_{@l}^* \rightsquigarrow_{r\beta} s'$ and $\underline{s}' = u$.*

Proof. Let $\text{nf}_{@l}(s)$ be the normal form of s with respect to $\rightsquigarrow_{@l}$, that exists and is unique by termination of $\rightsquigarrow_{@l}$ (Lemma 4.2.1) and determinism of the machine (Lemma 4.2.2). Since $\rightsquigarrow_{@l}$ is mapped on identities (Lemma 4.1.2) one has $\text{nf}_{@l}(s) = \underline{s}$. By hypothesis $\underline{s} \rightarrow_{wh}$ -reduces, so that by progress (Lemma 4.2.3) $\text{nf}_{@l}(s)$ cannot be final. Then $\text{nf}_{@l}(s) \rightsquigarrow_{r\beta} s'$, and $\underline{\text{nf}_{@l}(s)} = \underline{s} \rightarrow_{wh} \underline{s}'$ by the one-step simulation lemma (Lemma 4.1.1). By determinism of \rightarrow_{wh} , one obtains $\underline{s}' = u$. \square

Finally, we obtain the implementation theorem.

Theorem 4.4. *The Searching AM implements the weak head strategy.*

Proof. *Executions to Derivations:* by induction on the length $|\rho|$ of ρ using Lemma 4.1. *Derivations to Executions:* by induction on the length $|d|$ of d using Lemma 4.3 and noting that $\underline{t}^\circ = t$. \square

5 The Micro-Substituting Abstract Machine

Decomposing Meta-Level Substitution. The second task of abstract machines is to replace meta-level substitution $\bar{t}\{x \leftarrow \bar{u}\}$ with *micro-step substitution on demand*, i.e. a parsimonious approximation of meta-level substitution based on:

1. *Sharing:* when a β -redex $(\lambda x.\bar{t})\bar{u}$ is in evaluation position it is fired but the meta-level substitution $\bar{t}\{x \leftarrow \bar{u}\}$ is delayed, by introducing an annotation $[x \leftarrow \bar{u}]$ in a data-structure for delayed substitutions called *environment*;
2. *Micro-Step Substitution:* variable occurrences are replaced one at a time;
3. *Substitution on Demand:* replacement of a variable occurrence happens only when it ends up in evaluation position—variable occurrences that do not end in evaluation position are never substituted.

The purpose of this section is to illustrate this process in isolation via the study of a toy machine, the *Micro-Substituting Abstract Machine* (Micro AM) in Fig. 2, forgetting about the search for redexes.

Environments. We are going to treat environments in an unusual way: the literature mostly deals with *local* environments, to be discussed in Sect. 9, while here we prefer to first address the simpler notion of *global* environment, but to ease the terminology we will simply call them *environments*. So, an *environment* E is a list of entries of the form $[x \leftarrow \bar{u}]$. Each entry denotes the *delayed* substitution of \bar{u} for x . In a state $(\bar{t}, E' :: [x \leftarrow \bar{u}] :: E'')$ the scope of x is given by \bar{t} and E' , as it is stated by forthcoming Lemma 5.1. The (global) environment models a store. As it is standard in the literature, it is a *list*, but the list structure is only used to obtain a simple decoding and a handy delimitation of the scope of its entries. These properties are useful to develop the meta-theory of abstract machines, but keep in mind that (global) environments are not meant to be implemented as lists.

Code. The code under evaluation is now a λ -term $h\bar{r}_1 \dots \bar{r}_k$ expressed as a head h (that is either a β -redex $(\lambda x.\bar{t})\bar{u}$ or a variable x) applied to k arguments—it is a by-product of the fact that the Micro AM does not address *search*.

Transitions. There are two transitions:

- *Delaying β :* transition $\rightsquigarrow_{d\beta}$ removes the β -redex $(\lambda x.\bar{t})\bar{u}$ but does not execute the expected substitution $\{x \leftarrow \bar{u}\}$, it rather delays it, adding $[x \leftarrow \bar{u}]$ to the environment. It is the β -transition of the Micro AM.
- *Micro-Substitution On Demand:* if the head of the code is a variable x and there is an entry $[x \leftarrow \bar{t}]$ in the environment then transition \rightsquigarrow_{var} replaces that occurrence of x —and only that occurrence—with a copy of \bar{t} . It is necessary to rename the new copy of \bar{t} (into a well-named term) to avoid name clashes. It is the overhead transition of the Micro AM.

Implementation. Compilation sends a (well-named) term t to the initial state (\bar{t}, ε) , as for the Searching AM (but now the empty data-structure is the environment). The decoding simply applies the delayed substitutions in the environment to the term, considering them as meta-level substitutions.

The implementation of weak head reduction \rightarrow_{wh} by the Micro AM can be shown using the recipe given for the Searching AM, and it is therefore omitted. The only difference is in the proof that the overhead transition \rightsquigarrow_{var} terminates, that is based on a different argument. We spell it out because it will be useful also later on for complexity analyses. It requires the following invariant of machine executions:

Lemma 5.1 (Name Invariant). *Let $s = (\bar{t}, E)$ be a Micro AM reachable state.*

1. Abstractions: if $\lambda x.\bar{u}$ is a subterm of \bar{t} or of any code in E then x may occur only in \bar{u} ;
2. Environment: if $E = E' :: [x \leftarrow \bar{u}] :: E''$ then x is fresh with respect to \bar{u} and E'' .

Proof. By induction on the length of the execution ρ leading to s . If ρ is empty then s is initial and the statement holds because \bar{t} is well-named by hypothesis. If ρ is non-empty then it follows from the *i.h.* and the fact that transitions preserve the invariant, as an immediate inspection shows. \square

Lemma 5.2 (Micro-Substitution Terminates). *\rightsquigarrow_{var} terminates in at most $|E|$ steps (on reachable states).*

Proof. Consider a \rightsquigarrow_{var} transition copying \bar{u} from the environment $E' :: [x \leftarrow \bar{u}] :: E''$. If the next transition is again \rightsquigarrow_{var} , then the head of \bar{u} is a variable y and the transition copies from an entry in E'' because by Lemma 5.1 y cannot be bound by the entries in E' . Then the number of consecutive \rightsquigarrow_{var} transitions is bound by $|E|$ (that is not extended by \rightsquigarrow_{var}). \square

Theorem 5.3. *The Micro AM implements the weak head strategy.*

$$\begin{array}{ll}
 \text{Environments} & E := \varepsilon \mid [x \leftarrow \bar{t}] :: E \\
 \text{Stacks} & \pi := \varepsilon \mid \bar{t} :: \pi \\
 \text{Compilation} & t^\circ := (\bar{t}, \varepsilon, \varepsilon)
 \end{array}
 \quad \mid \quad
 \begin{array}{ll}
 \text{Decoding} & \frac{(\bar{t}, \varepsilon, \varepsilon)}{(\bar{t}, \bar{u} :: \pi, E)} := t \\
 & \frac{(\bar{t}, \bar{u} :: \pi, E)}{(\bar{t}, \varepsilon, [x \leftarrow \bar{u}] :: E)} := \frac{(\bar{t}\bar{u}, \pi, E)}{(\bar{t}\{x \leftarrow \bar{u}\}, \varepsilon, E)}
 \end{array}$$

Code	Stack	Env	Trans	Code	Stack	Env
$\bar{t}\bar{u}$	π	E	$\rightsquigarrow_{@l}$	\bar{t}	$\bar{u} :: \pi$	E
$\lambda x.\bar{t}$	$\bar{u} :: \pi$	E	$\rightsquigarrow_{r\beta}$	\bar{t}	π	$[x \leftarrow \bar{u}] :: E$
x	π	$E :: [x \leftarrow \bar{t}] :: E'$	\rightsquigarrow_{var}	\bar{t}^α	π	$E :: [x \leftarrow \bar{t}] :: E'$

where \bar{t}^α denotes \bar{t} where bound names have been freshly renamed.

Figure 3: Milner Abstract Machine (MAM).

6 Search + Micro-Substitution = Milner Abstract Machine

The Searching AM and the Micro AM can be merged together into the Milner Abstract Machine (MAM), defined in Fig. 3. The MAM has both an argument stack and an environment. The machine has one β -transition $\rightsquigarrow_{r\beta}$ inherited from the Searching AM, and two overhead transitions, $\rightsquigarrow_{@l}$ inherited from the the Searching AM and \rightsquigarrow_{var} inherited from the Micro AM. Note that in \rightsquigarrow_{var} the code now is simply a variable, because the arguments are supposed to be stored in the argument stack.

For the implementation theorem once again the only delicate point is to prove that the overhead transitions terminate. As for the Micro AM one needs a name invariant. A termination measure can then be defined easily by mixing the size of the codes (needed for $\rightsquigarrow_{@l}$) and the size of the environment (needed for \rightsquigarrow_{var}), and it is omitted here, because it will be exhaustively studied for the complexity analysis of the MAM. Therefore, we obtain that:

Theorem 6.1. *The MAM implements the weak head strategy.*

7 Introducing Complexity Analyses

The complexity analysis of abstract machines is the study of the asymptotic behavior of their overhead.

Parameters for Complexity Analyses. Let us reason abstractly, by considering a generic strategy \rightarrow in the λ -calculus and a given machine M implementing \rightarrow . By the *derivations-to-executions* part of the implementation (Definition 3.1), given a derivation $d : t_0 \rightarrow^n u$ there is a shortest execution $\rho : t_0^\circ \rightsquigarrow_M s$ such that $s = u$. Determining the complexity of M amounts to bound the complexity of a concrete implementation of ρ , say on a RAM model, as a function of two fundamental parameters:

1. *Input:* the size $|t_0|$ of the initial term t_0 of the derivation d ;
2. *Strategy* the length $n = |d|$ of the derivation d , that coincides with the number $|\rho|_\beta$ of β -transitions in ρ by the β -matching requirement for implementations.

Note that our notion of implementation allows to forget about the strategy while studying the complexity of the machine, because the two fundamental parameters are internalized: the input is simply the initial code and the length of the strategy is simply the number of β -transitions.

Types of Machines. The bound on the overhead of the machine is then used to classify it, as follows.

Definition 7.1. Let M an abstract machine implementing a strategy \rightarrow . Then

- M is reasonable if the complexity of M is polynomial in the input $|t_0|$ and the strategy $|\rho|_\beta$;
- M is unreasonable if it is not reasonable;
- M is efficient if it is linear in both the input and the strategy (we sometimes say that it is bilinear).

Recipe for Complexity Analyses. The estimation of the complexity of a machine usually takes 3 steps:

1. *Number of Transitions*: bound the length of the execution ρ simulating the derivation d , usually having a bound on every kind of transition of M .
2. *Cost of Single Transitions*: bound the cost of concretely implementing a single transition of M —different kind of transitions usually have different costs. Here it is usually necessary to go beyond the abstract level, making some (high-level) assumption on how codes and data-structure are concretely represented (our case study will provide examples).
3. *Complexity of the Overhead*: obtain the total bound by composing the first two points, that is, by taking the number of each kind of transition times the cost of implementing it, and summing over all kinds of transitions.

8 The Complexity of the MAM

In this section we provide the complexity analysis of the MAM, from which analyses of the Searching and Micro AM easily follow.

The Crucial Subterm Invariant. The analysis is based on the following subterm invariant.

Lemma 8.1 (Subterm Invariant). *Let $\rho : t_0^\circ \rightsquigarrow_{MAM} (\bar{u}, \pi, E)$ be a MAM execution. Then \bar{u} and any code in π and E are subterms of t_0 .*

Note that the MAM copies code only in transition \rightsquigarrow_{var} , where it copies a code from the environment E . Therefore, the subterm invariant bounds the size of the subterms duplicated along the execution.

Let us be precise about *subterms*: for us, \bar{u} is a subterm of t_0 if it does so up to variable names, both free and bound (and so the distinction between terms and codes is irrelevant). More precisely: define t^- as t in which all variables (including those appearing in binders) are replaced by a fixed symbol $*$. Then, we will consider u to be a subterm of t whenever u^- is a subterm of t^- in the usual sense. The key property ensured by this definition is that the size $|\bar{u}|$ of \bar{u} is bounded by $|\bar{t}|$.

Proof. By induction on the length of ρ . The base case is immediate and the inductive one follows from the *i.h.* and the immediate fact that the transitions preserve the invariant. \square

The subterm invariant is crucial, for two related reasons. First, it linearly relates the cost of duplications to the size of the input, enabling complexity analyses. With respect to the length of the strategy, then, micro-step operations have constant cost, as required by the recipe for micro-step operational semantics in Sect. 2. Second, it implies that size-explosion has been circumvented: duplications are linear, and so the size of the state can grow at most linearly with the number of steps, *i.e.* it cannot explode. In particular, we also obtain the compact representation of the results required by the recipe.

The relevance of the subterm invariant goes in fact well beyond abstract machines, as it is typical of most instances of micro-step operational semantics. And for complexity analyses of the λ -calculus it is absolutely essential, playing a role analogous to that of the cut-elimination theorem in the study of sequent calculi or of the sub-formula property for proof search.

Number of Transitions. The next lemma bounds the global number of overhead transitions. For the micro-substituting transition \rightsquigarrow_{var} it relies on an auxiliary bound of a more local form. For the searching transition $\rightsquigarrow_{@l}$ the bound relies on the subterm invariant. We denote with $|\rho|_\beta$, $|\rho|_{@l}$, and $|\rho|_{var}$ the number of $\rightsquigarrow_{r\beta}$, $\rightsquigarrow_{@l}$, and \rightsquigarrow_{var} transitions in ρ , respectively.

Lemma 8.2. *Let $\rho : t_0^\circ \rightsquigarrow_{MAM} s$ be a MAM execution. Then:*

1. Micro-Substitution Linear Local Bound: if $\sigma : s \rightsquigarrow_{@l, var}^* s'$ then $|\sigma|_{var} \leq |E| = |\rho|_\beta$;
2. Micro-Substitution Quadratic Global Bound: $|\rho|_{var} \leq |\rho|_\beta^2$;
3. Searching (and β) Local Bound: if $\sigma : s \rightsquigarrow_{r\beta, @l}^* s'$ then $|\sigma| \leq |t_0|$;
4. Searching Global Bound: $|\rho|_{@l} \leq |t_0| \cdot (|\rho|_{var} + 1) \leq |t_0| \cdot (|\rho|_\beta^2 + 1)$.

Proof.

1. Reasoning along the lines of Lemma 5.2 one obtains that \rightsquigarrow_{var} transitions in σ have to use entries of E from left to right ($\rightsquigarrow_{@l}$ and \rightsquigarrow_{var} do not modify E), and so $|\sigma|_{var} \leq |E|$. Now, $|E|$ is exactly $|\rho|_\beta$, because the only transition extending E , and of exactly one entry, is $\rightsquigarrow_{r\beta}$.
2. The fact that a linear local bound induces a quadratic global bound is a standard reasoning. We spell it out to help the unacquainted reader. The execution ρ alternates phases of β -transitions and phases of overhead transitions, *i.e.* it has the shape:

$$t_0^\circ = s_1 \rightsquigarrow_{r\beta}^* s'_1 \rightsquigarrow_{@l, var}^* s_2 \rightsquigarrow_{r\beta}^* s'_2 \rightsquigarrow_{@l, var}^* \dots s_k \rightsquigarrow_{r\beta}^* s'_k \rightsquigarrow_{@l, var}^* s$$

Let a_i be the length of the segment $s_i \rightsquigarrow_{r\beta}^* s'_i$ and b_i be the number of \rightsquigarrow_{var} transitions in the segment $s'_i \rightsquigarrow_{@l, var}^* s_{i+1}$, for $i = 1, \dots, k$. By Point 1, we obtain $b_i \leq \sum_{j=1}^i a_j$. Then $|\rho|_{var} = \sum_{i=1}^k b_i \leq \sum_{i=1}^k \sum_{j=1}^i a_j \leq \sum_{j=1}^k a_j = |\rho|_\beta$ and $k \leq |\rho|_\beta$. So $|\rho|_{var} \leq \sum_{i=1}^k \sum_{j=1}^i a_j \leq \sum_{i=1}^k |\rho|_\beta \leq |\rho|_\beta^2$.

3. The length of σ is bound by the size of the code in the state s because $\rightsquigarrow_{r\beta, @l}$ strictly decreases the size of the code, that in turn is bound by the size $|t_0|$ of the initial term by the subterm invariant (Lemma 8.1).
4. The execution ρ alternates phases of $\rightsquigarrow_{r\beta}$ and $\rightsquigarrow_{@l}$ transitions and phases of \rightsquigarrow_{var} transitions, *i.e.* it has the shape:

$$t_0^\circ = s_1 \rightsquigarrow_{r\beta, @l}^* s'_1 \rightsquigarrow_{var}^* s_2 \rightsquigarrow_{r\beta, @l}^* s'_2 \rightsquigarrow_{var}^* \dots s_k \rightsquigarrow_{r\beta, @l}^* s'_k \rightsquigarrow_{var}^* \rightsquigarrow_{r\beta, @l}^* s$$

By Point 3 the length of the segments $s_i \rightsquigarrow_{r\beta, @l}^* s'_i$ is bound by the size $|t_0|$ of the initial term. The code may grow, instead, with \rightsquigarrow_{var} transitions. So $|\rho|_{@l}$ is bound by $|t_0|$ times the number $|\rho|_{var}$ of micro-substitution transitions, plus $|t_0|$ once more, because at the beginning there might be $\rightsquigarrow_{r\beta, @l}$ transitions before any \rightsquigarrow_{var} transition—in symbols, $|\rho|_{@l} \leq |t_0| \cdot (|\rho|_{var} + 1)$. Finally, $|t_0| \cdot (|\rho|_{var} + 1) \leq |t_0| \cdot (|\rho|_\beta^2 + 1)$ by Point 2. \square

Cost of Single Transitions. To estimate the cost of concretely implementing single transitions we need to make some hypotheses on how the MAM is going to be itself implemented on RAM:

1. *Codes, Variable (Occurrences), and Environment Entries:* abstractions and applications are constructors with pointers to subterms, a variable is a memory location, a variable occurrence is a reference to that location, and an environment entry $[x \leftarrow \bar{t}]$ is the fact that the location associated to x contains (the topmost constructor of) \bar{t} .
2. *Random Access to Global Environments:* the environment E of the MAM can be accessed in constant time (in \rightsquigarrow_{var}) by just following the reference given by the variable occurrence under evaluation, with no need to access E sequentially, thus ignoring its list structure.

It is now possible to bound the cost of single transitions. Note that the case of \rightsquigarrow_{var} transitions relies on the subterm invariant.

Lemma 8.3. *Let $\rho : t_0^\circ \rightsquigarrow_{MAM} s$ be a MAM execution. Then:*

1. *Each $\rightsquigarrow_{@l}$ transition in ρ has constant cost;*
2. *Each $\rightsquigarrow_{r\beta}$ transition in ρ has constant cost;*
3. *Each \rightsquigarrow_{var} transition in ρ has cost bounded by the size $|t_0|$ of the initial term.*

Proof. According to our hypothesis on the concrete implementation of the MAM, $\rightsquigarrow_{@l}$ just moves the pointer to the current code on the left subterm of the application and pushes the pointer to the right subterm on the stack—evidently constant time. Similarly for $\rightsquigarrow_{r\beta}$. For \rightsquigarrow_{var} , the environment entry $[x \leftarrow \bar{t}]$ is accessed in constant time by hypothesis, but \bar{t} has to be α -renamed, *i.e.* copied. It is not hard to see that this can be done in time linear in $|\bar{t}|$ (the naive algorithm for copying carries around a list of variables, and it is quadratic, but it can be easily improved to be linear) that by the subterm invariant (Lemma 8.1) is bound by the size $|t_0|$ of the initial term. \square

Complexity of the Overhead. By composing the analysis of the number of transitions (Lemma 8.2) with the analysis of the cost of single transitions (Lemma 8.3) we obtain the main result of the paper.

Theorem 8.4 (The MAM is Reasonable). *Let $\rho : t_0^\circ \rightsquigarrow_{MAM} s$ be a MAM execution. Then:*

1. $\rightsquigarrow_{@l}$ transitions in ρ cost all together $O(|t_0| \cdot (|\rho|_\beta^2 + 1))$;
2. $\rightsquigarrow_{r\beta}$ transitions in ρ cost all together $O(|\rho|_\beta)$;
3. \rightsquigarrow_{var} transitions in ρ cost all together $O(|t_0| \cdot (|\rho|_\beta^2 + 1))$;

Then ρ can be implemented on RAM with cost $O(|t_0| \cdot (|\rho|_\beta^2 + 1))$, *i.e.* the MAM is a reasonable implementation of the weak head strategy.

The Efficient MAM. According to the terminology of Sect. 3, the MAM is reasonable but it is not efficient because micro-substitution takes time quadratic in the length of the strategy. The quadratic factor comes from the fact that in the environment there can be growing chains of renamings, *i.e.* of substitutions of variables for variables, see [6] for more details on this issue. The MAM can actually be optimized easily, obtaining an efficient implementation, by replacing $\rightsquigarrow_{r\beta}$ with the following two β -transitions:

$\lambda x.\bar{t}$	$y :: \pi$	E	$\rightsquigarrow_{r\beta_1}$	$\bar{t}\{x \leftarrow y\}$	π	E
$\lambda x.\bar{t}$	$\bar{u} :: \pi$	E	$\rightsquigarrow_{r\beta_2}$	\bar{t}	π	$[x \leftarrow \bar{u}] :: E$

if \bar{u} is not a variable

Search is Linear and the Micro AM is Reasonable. By Lemma 8.2 the cost of search in the MAM is linear in the number of transitions for implementing micro-substitution. This is an instance of a more general fact: *search* turns out to always be bilinear (in the initial code and in the amount of micro-substitutions). There are two consequences of this general fact. First, it can be turned into a design principle for abstract machines—*search has to be bilinear*, otherwise there is something wrong in the design of the machine. Second, search is somewhat negligible for complexity analyses.

The Micro AM can be seen as the MAM up to search. In particular, it satisfies a subterm invariant and thus circumvents size-explosion. The Micro AM is however quite less efficient, because at each step it has to search the redex from scratch. An easy but omitted analysis shows that its overhead is nonetheless polynomial. Therefore, it makes sense to consider very abstract machines as the Micro AM that omit search. In fact, they already exist, in disguise, as strategies in the *linear substitution calculus* [1, 5], a recent approach to explicit substitutions modeling exactly micro-substitution without search (the traditional approach to explicit substitutions instead models both micro-substitution and search) and they were used for the first proof that a strong strategy (the leftmost-outermost one) is reasonable [10].

The Searching AM is Unreasonable. It is not hard to see that the Searching AM is unreasonable. Actually, the number of transitions is reasonable. The projection of MAM executions on Searching AM executions, indeed, shows that the number of searching transitions of the Searching AM is reasonable. It is the cost of single transitions that becomes unreasonable. In fact, the Searching AM does not have a subterm invariant, because it rests on meta-level substitution, and the size of the terms duplicated by the $\rightsquigarrow_{r\beta}$ transition can explode (it is enough to consider the size-exploding family of Proposition 1.2).

The moral is that micro-substitution is more fundamental than search. While the cost of search can be expressed in terms of the cost of micro-substitution, the converse is in fact not possible.

9 Names: Krivine Abstract Machine

Accounting for Names. In the study presented so far we repeatedly took names seriously, by distinguishing between terms and codes, by asking that initial codes are well-named, and by proving an invariant about names (Lemma 5.1). The process of α -renaming however has not been made explicit, the machines we presented rather rely on a meta-level renaming, used as a black box.

The majority of the literature on abstract machines, instead, pays more attention to α -equivalence, or rather to how to avoid it. We distinguish two levels:

1. *Removal of on-the-fly α -equivalence*: in these cases the machine works on terms with variable names but it is designed in order to implement evaluation without ever α -renaming. Technically, the global environment of the MAM is replaced by many local environments, each one for every piece of code in the machine. The machine becomes more complex, in particular the non-trivial concept of closure (to be introduced shortly) is necessary.
2. *Removal of names*: terms are represented using de Bruijn indexes (or de Bruijn levels), removing the problem of α -equivalence altogether but sacrificing the readability of the machine and reducing its abstract character. Usually this level is built on top of the previous one.

We are now going to introduce Krivine Abstract Machine (keeping names, so at the first level), yet another implementation of the weak head strategy. Essentially, it is a version of the MAM without on-the-fly α -equivalence. The complexity analysis will show that it has exactly the same complexity of the MAM. The further removal of names is only (anti)cosmetic—the complexity is not affected either. Consequently, the task of accounting for names is—as for search—negligible for complexity analyses.

Local Env. $e := \varepsilon \mid [x \leftarrow c] :: e$ Closures $e := (\bar{t}, e)$ Stacks $\pi := \varepsilon \mid c :: \pi$ States $s := (c, \pi)$ Compilation $t^\circ := ((\bar{t}, \varepsilon), \varepsilon)$	Closure Decoding $\frac{(\bar{t}, \varepsilon)}{(\bar{t}, [x \leftarrow c] :: e)} := t$ State Decoding $\frac{(\bar{t}, [x \leftarrow c] :: e)}{\frac{(c, \varepsilon)}{(c, c' :: \pi)}} := \frac{(\bar{t}\{x \leftarrow c\}, e)}{c}$ $\frac{(c, c' :: \pi)}{((cc', \varepsilon), \pi)} := ((cc', \varepsilon), \pi)$																												
<table border="1" style="margin: auto; border-collapse: collapse;"> <thead> <tr> <th><i>Code</i></th><th><i>LocEnv</i></th><th><i>Stack</i></th><th><i>Trans</i></th><th><i>Code</i></th><th><i>LocEnv</i></th><th><i>Stack</i></th></tr> </thead> <tbody> <tr> <td>$\bar{t}u$</td><td>e</td><td>π</td><td>$\rightsquigarrow_{@l}$</td><td>\bar{t}</td><td>e</td><td>$(\bar{u}, e) :: \pi$</td></tr> <tr> <td>$\lambda x.\bar{t}$</td><td>e</td><td>$c :: \pi$</td><td>$\rightsquigarrow_{r\beta}$</td><td>$\bar{t}$</td><td>$[x \leftarrow c] :: e$</td><td>$\pi$</td></tr> <tr> <td>$x$</td><td>$e$</td><td>$\pi$</td><td>$\rightsquigarrow_{var}$</td><td>$\bar{t}$</td><td>$e'$</td><td>$\pi$</td></tr> </tbody> </table>	<i>Code</i>	<i>LocEnv</i>	<i>Stack</i>	<i>Trans</i>	<i>Code</i>	<i>LocEnv</i>	<i>Stack</i>	$\bar{t}u$	e	π	$\rightsquigarrow_{@l}$	\bar{t}	e	$(\bar{u}, e) :: \pi$	$\lambda x.\bar{t}$	e	$c :: \pi$	$\rightsquigarrow_{r\beta}$	\bar{t}	$[x \leftarrow c] :: e$	π	x	e	π	\rightsquigarrow_{var}	\bar{t}	e'	π	with $e(x) = (\bar{t}, e')$
<i>Code</i>	<i>LocEnv</i>	<i>Stack</i>	<i>Trans</i>	<i>Code</i>	<i>LocEnv</i>	<i>Stack</i>																							
$\bar{t}u$	e	π	$\rightsquigarrow_{@l}$	\bar{t}	e	$(\bar{u}, e) :: \pi$																							
$\lambda x.\bar{t}$	e	$c :: \pi$	$\rightsquigarrow_{r\beta}$	\bar{t}	$[x \leftarrow c] :: e$	π																							
x	e	π	\rightsquigarrow_{var}	\bar{t}	e'	π																							

Figure 4: Krivine Abstract Machine (KAM).

Krivine Abstract Machine. The machine is in Fig. 4. It relies on the mutually inductively defined concepts of *local environment*, that is a list of closures, and *closure*, that is a list of pairs of a code and a local environment. A state is a *pair* of a closure and a stack, but in the description of the transitions we write it as a *tuple*, by spelling out the two components of the closure. Let us explain the name *closure*: usually, machines are executed on closed terms, and then a closure decodes indeed to a closed term. While essential in the study of call-by-value or call-by-need strategies, for the weak head (call-by-name) strategy the closed hypothesis is unnecessary, that is why we do not deal with it—so a closure here does not necessarily decode to a closed term. Two remarks:

1. *Garbage Collection*: transition \rightsquigarrow_{var} , beyond implementing micro-substitution, also accounts for some garbage collection, as it throws away the local environment e associated to the replaced variable x . The MAM simply ignores garbage collection. For time analyses garbage collection can indeed be safely ignored, while it is clearly essential for space (both the KAM and the MAM are however desperately inefficient with respect to space).
2. *No α -Renaming and the Length of Local Environments*: names are never renamed. The initial code, as usual, is assumed to be well-named. Then the entries of a same local environment are all on distinguished names (formally, a name invariant holds). Then the length of a local environment e is bound by the number of names in the initial term, that is, by the size of the initial term (formally, $|e| \leq |t_0|$). This essential quantitative invariant is used in analysis of the next paragraph.

Implementation and Complexity Analysis. The proof that the KAM implements the weak head strategy follows the recipe for these proofs and it is omitted. For the complexity analysis, the bound of the number of transitions can be shown to be exactly as for the MAM. A direct proof is not so simple, because the bound on \rightsquigarrow_{var} transitions cannot exploit the size of the global environment. The bound can be obtained by relating the KAM with the Searching AM (for which the exact same bound of the MAM holds), or by considering the *depth* (*i.e.* maximum nesting) of local environments. The proof is omitted.

The interesting part of the analysis is rather the study of the cost of single transitions. As for the MAM, we need to spell out the hypotheses on how the KAM is concretely implemented on RAM. Variables cannot be implemented with pointers, because the same variable name can be associated to different codes in different local environments. So they have to simply be numbers. Then there are two choices for the representation of environments, either they are represented as lists or as arrays. In both cases $\rightsquigarrow_{r\beta}$ can be implemented in constant time. For the other transitions:

1. *Environments as Arrays*: we mentioned that there is a bound on the length of local environments ($|e| \leq |t_0|$) so that arrays can be used. The choice allows to implement \rightsquigarrow_{var} in constant time, because e can be accessed directly at the position described by the number given by x . Transition $\rightsquigarrow_{@l}$ however requires to duplicate e , and this is necessary because the two copies might later on be modified differently. So the cost of a $\rightsquigarrow_{@l}$ transition becomes linear in $|t_0|$ and $\rightsquigarrow_{@l}$ transitions all together cost $O(|t_0|^2 \cdot (|\rho|_\beta^2 + 1))$, that also becomes the complexity of the whole overhead of the KAM. This is worse than the MAM.
2. *Environments as Lists*: implementing local environments as lists provides sharing of environments, overcoming the problems of arrays. With lists, transition $\rightsquigarrow_{@l}$ becomes constant time, as for the MAM, because the copy of e now is simply the copy of a pointer. The trick is that the two copies of the environment can only be extended differently *on the head*, so that the tail of the list can be shared. Transition \rightsquigarrow_{var} however now needs to access e sequentially, and so it costs $|t_0|$ as for the MAM. Thus globally we obtain the same overhead of the MAM.

Summing up, *names* can be pushed at the meta-level (as in the MAM) without affecting the complexity of the overhead. Thus, *names* are even less relevant than *search* at the level of complexity. The moral of this tutorial then is that *substitution* is the crucial aspect for the complexity of abstract machines.

References

- [1] Beniamino Accattoli (2012): *An Abstract Factorization Theorem for Explicit Substitutions*. In: RTA, pp. 6–21. Available at <http://dx.doi.org/10.4230/LIPIcs.RTA.2012.6>.
- [2] Beniamino Accattoli (2016): *The Useful MAM, a Reasonable Implementation of the Strong λ -Calculus*. In: WoLLIC 2016, pp. 1–21. Available at http://dx.doi.org/10.1007/978-3-662-52921-8_1.
- [3] Beniamino Accattoli, Pablo Barenbaum & Damiano Mazza (2014): *Distilling abstract machines*. In: ICFP 2014, pp. 363–376. Available at <http://doi.acm.org/10.1145/2628136.2628154>.
- [4] Beniamino Accattoli, Pablo Barenbaum & Damiano Mazza (2015): *A Strong Distillery*. In: APLAS 2015, pp. 231–250. Available at http://dx.doi.org/10.1007/978-3-319-26529-2_13.
- [5] Beniamino Accattoli, Eduardo Bonelli, Delia Kesner & Carlos Lombardi (2014): *A nonstandard standardization theorem*. In: POPL, pp. 659–670. Available at <http://doi.acm.org/10.1145/2535838.2535886>.
- [6] Beniamino Accattoli & Claudio Sacerdoti Coen (2014): *On the Value of Variables*. In: WoLLIC 2014, pp. 36–50. Available at http://dx.doi.org/10.1007/978-3-662-44145-9_3.
- [7] Beniamino Accattoli & Claudio Sacerdoti Coen (2015): *On the Relative Usefulness of Fireballs*. In: LICS 2015, pp. 141–155. Available at <http://dx.doi.org/10.1109/LICS.2015.23>.
- [8] Beniamino Accattoli & Ugo Dal Lago (2012): *On the Invariance of the Unitary Cost Model for Head Reduction*. In: RTA, pp. 22–37. Available at <http://dx.doi.org/10.4230/LIPIcs.RTA.2012.22>.
- [9] Beniamino Accattoli & Delia Kesner (2010): *The Structural λ -Calculus*. In: CSL, pp. 381–395. Available at http://dx.doi.org/10.1007/978-3-642-15205-4_30.
- [10] Beniamino Accattoli & Ugo Dal Lago (2014): *Beta reduction is invariant, indeed*. In: CSL-LICS ’14, pp. 8:1–8:10. Available at <http://doi.acm.org/10.1145/2603088.2603105>.
- [11] Guy E. Blelloch & John Greiner (1995): *Parallelism in Sequential Functional Languages*. In: FPCA, pp. 226–237. Available at <http://doi.acm.org/10.1145/224164.224210>.
- [12] Ugo Dal Lago & Simone Martini (2006): *An Invariant Cost Model for the Lambda Calculus*. In: CiE 2006, pp. 105–114. Available at http://dx.doi.org/10.1007/11780342_11.
- [13] David Sands, Jörgen Gustavsson & Andrew Moran (2002): *Lambda Calculi and Linear Speedups*. In: The Essence of Computation, pp. 60–84. Available at http://dx.doi.org/10.1007/3-540-36377-7_4.